Name

erf, erfc – error functions

Syntax

#include <math.h>

double erf(*x***)**
double *x***;**

double erfc(*x***)**
double *x***;**

Description

The `erf` function returns the error function of *x* defined as follows:

```
erf(x) = 2/sqrt(pi)*integral from 0 to x of exp(-t*t) dt.
```

The `erfc` function returns 1.0–erf(*x*).

The entry for the `erfc` function is provided because of the extreme loss of relative accuracy if erf(*x*) is called for large *x* and the result subtracted from 1. For example if *x* = 10, 12 places are lost.

Return Value

The `erf` and `erfc` functions return *NaN* when *x* is *NaN*.

See Also

math(3m)

Name

erf, erfc – error function and complementary error function

Syntax

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

Description

The `erf` function returns the error function of x, defined as $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

The `erfc` function, which returns 1.0 – *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0 (e.g. for $x = 5$, 12 places are lost).

See Also

exp(3m)
ULTRIX Programmer's Manual, Unsupported

Name

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power

Syntax

```
#include <math.h>

double exp(x)
double x;

float fexp(x)
float x;

double expm1(x)
double x;

float fexpm1(x)
float x;

double log(x)
double x;

float flog(x)
float x;

double log10(x)
double x;

float flog10(x)
float x;

double log1p(x)
double x;

float flog1p(x)
float x;

double pow(x,y)
double x,y;
```

Description

The `exp` and `fexp` functions return the exponential function of *x* for double and float data types, respectively.

The `expm1` and `fexpm1` functions return exp(*x*)-1 accurately, including tiny *x* for double and float data types, respectively.

The `log` and `flog` functions return the natural logarithm of *x* for double and float data types, respectively.

The `log10` and `flog10` functions return the logarithm of *x* to base 10 for double and float data types, respectively.

The `log1p` and `flog1p` functions return log(1+*x*) accurately, including tiny *x* for double and float data types, respectively.

The `pow` function returns *x****y*.

Error (due to roundoff)

The `exp`, `log`, `expm1`, and `log1p` functions are accurate to within an *ulp*, and `log10` is accurate to within approximately 2 *ulp*s; an *ulp* is one *U*nit in the *L*ast *P*lace.

The `pow` function is accurate to within 2 *ulp*s when its magnitude is moderate, but becomes less accurate as the `pow` result approaches the overflow or underflow thresholds. Theoretically, as these thresholds are approached, almost as many bits could be lost from the result as are indicated in the exponent field of the floating-point format for the resultant number. In other words, up to 11 bits for an IEEE 754 double-precision floating-point number. However, testing has never verified loss of precision as drastic as 11 bits. The worst cases have shown accuracy of results to within 300 *ulp*s for IEEE 754 double-precision floating-point numbers. In general, a `pow` (integer, integer) result is exact until it is larger than 2**53 (for IEEE 754 double-precision floating-point).

Return Value

All of the double precision functions return *NaN* if *x* or *y* is *NaN*.

The `exp` function returns HUGE_VAL when the correct value would overflow, and zero when the correct value would underflow.

The `log` and `log10` functions return *NaN* when *x* is less than or equal to zero or when the correct value would overflow.

The `pow` function returns *NaN* if *x* or *y* is *NaN*. When both *x* and *y* are zero, 1.0 is returned. When *x* is negative and *y* is not an integer, *NaN* is returned. If *x* is zero and *y* is negative, -HUGE_VAL is returned.

The `sqrt` function returns *NaN* when *x* is negative.

See Also

math(3m)

Name

exp, expm1, log, log10, log1p, pow, sqrt – exponential, logarithm, power, square root

Syntax

#include <math.h>

double exp(*x***)**
double *x***;**

double expm1(*x***)**
double *x***;**

double log(*x***)**
double *x***;**

double log10(*x***)**
double *x***;**

double log1p(*x***)**
double *x***;**

double pow(*x,y***)**
double *x,y***;**

double sqrt(*x***)**
double *x***;**

Description

The `exp` function returns the exponential function of *x*.

The `expm1` function returns exp(*x*)-1 accurately even for tiny *x*.

The `log` function returns the natural logarithm of *x*; `log10` returns the base 10 logarithm.

The `log1p` function returns log(1+*x*) accurately even for tiny *x*.

The `pow` function returns *x* raised to the *y* power.

The `sqrt` function returns the square root of *x*.

Return Value

The `exp` function returns HUGE_VAL and sets *errno* to ERANGE when the correct value would overflow. When the correct value would underflow it returns zero and *errno* is set to ERANGE.

The `expm1` function returns HUGE_VAL and sets *errno* to ERANGE when the correct value would overflow. When the correct value would underflow it returns -1.

The `log` and `log10` functions return -HUGE_VAL and set *errno* to EDOM when *x* is less than or equal to zero. When the correct value would overflow flow they return -HUGE_VAL and *errno* is set to ERANGE.

The `log1p` function returns -HUGE_VAL and sets *errno* to EDOM when *x* is less than or equal to -1. When the correct value would overflow flow it returns -HUGE_VAL and *errno* is set to ERANGE.

The `pow` function has many special cases. When *x* and *y* are both zero it returns 1.0. When *x* is negative and *y* is not an integer value it returns zero and *errno* is set to EDOM. When *x* is zero and *y* is negative it returns -HUGE_VAL and *errno* is set to EDOM. When the correct value would overflow HUGE_VAL is returned and *errno* is set to ERANGE. When the correct value would underflow zero is returned and *errno* is set to ERANGE.

The `sqrt` function returns zero and sets *errno* to EDOM when *x* is negative.

Environment

When your program is compiled using the System V environment, `exp` returns HUGE when the correct value would overflow, and sets *errno* to ERANGE; `exp` returns zero when the correct value would underflow, and sets errno to ERANGE.

The `log` and `log10` functions return HUGE and set *errno* to EDOM when *x* is nonpositive. An error message is printed on the standard error output.

The `pow` function returns zero and sets *errno* to EDOM when *x* is non-positive and y is not an integer, or when *x* and *y* are both zero. In these cases, a message indicating DOMAIN error is printed on the standard error output. When the correct value for `pow` would overflow, `pow` returns HUGE and sets *errno* to ERANGE.

The `sqrt` function returns zero and sets *errno* to EDOM when *x* is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function `matherr`(3m).

NOTE

DOMAIN error is only printed in the System V environment.

See Also

hypot(3m), intro(3m), sinh(3m)

Name

floor, ffloor, fabs, ceil, ceil, trunc, ftrunc, fmod, rint – floor, absolute value, ceiling, truncation, floating point remainder and round-to-nearest functions

Syntax

#include <math.h>

double floor(*x***)**
double *x***;**

float ffloor(*x***)**
float *x***;**

double ceil(*x***)**
double *x***;**

float fceil(*x***)**
float *x***;**

double trunc(*x***)**
double *x***;**

float ftrunc(*x***)**
float *x***;**

double fabs(*x***)**
double *x***;**

double fmod (*x***,** *y***)**
double *x***,** *y***;**

double rint(*x***)**
double *x***;**

Description

The `floor` and `ffloor` routines return the largest integer which is not greater than x for double and float data types, respectively.

The `ceil` and `fceil` routines return the smallest integer which is not less than x for double and float data types, respectively.

The `trunc` and `ftrunc` routines return the integer (represented as a floating-point number) of x with the fractional bits truncated for double and float data types respectively.

The `fabs` routine returns the absolute value |x|.

The `fmod` routine returns the floating point remainder of the division of x by y: zero if y is zero or if x/y would overflow; otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

The `rint` routine returns the integer (represented as a double precision number) nearest x in the direction of the prevailing rounding mode.

In the default rounding mode, to nearest, `rint(x)` is the integer nearest x with the additional stipulation that if |rint(x)–x|=1/2 then `rint(x)` is even. Other rounding modes can make `rint` act like `floor` or `ceil`, or round towards zero.

Another way to obtain an integer near x is to declare (in C)

```
double x;    int k;    k = x;
```

The C compiler rounds x towards 0 to get the integer k. Also note that, if x is larger than k can accommodate, the value of k and the presence or absence of an integer overflow are hard to predict.

The `fabs` routine is in libc.a rather than libm.a.

See Also

abs(3), ieee(3m), math(3m)

Name

fabs, floor, ceil, fmod, rint – absolute value, floor, ceiling, floating point remainder, and round-to-nearest functions

Syntax

#include <math.h>

double floor(*x*)
double *x*;

double ceil(*x*)
double *x*;

double fabs(*x*)
double *x*;

double fmod (*x, y*)
double *x, y*;

double rint(*x*)
double *x*;

Description

The `fabs` routine returns the absolute value |x|.

The `floor` routine returns the largest integer no greater than x.

The `ceil` routine returns the smallest integer no less than x.

The `fmod` routine returns the floating point remainder of the division of x by y: zero if y is zero or if x/y would overflow; otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

The `rint` routine returns the integer (represented as a double precision number) nearest x in the direction of the prevailing rounding mode.

See Also

abs(3), intro(3m)

Name

gamma, lgamma, signgam – log gamma function

Syntax

#include <math.h>

double gamma(*x***)**
double *x***;**

double lgamma(*x***)**
double *x***;**

extern int `signgam;`

Description

The `gamma` function returns ln $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer `signgam`. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0)
        error();
y = exp(y);
if(signgam)
        y = -y;
```

The `lgamma` function is another name for the `gamma` function.

Return Value

The `gamma` and `lgamma` functions return *NaN* when *x* is *NaN* or when it is an integer value less than or equal to zero. On overflow `gamma` and `lgamma` functions return HUGE_VAL.

Environment

When your program is compiled using the System V environment for nonpositive integer values, HUGE is returned, and *errno* is set to EDOM. A message indicating DOMAIN error is printed on the standard error output.

If the correct value would overflow, `gamma` returns HUGE and sets *errno* to ERANGE.

These error-handling procedures may be changed with the function `matherr`(3m).

See Also

matherr(3m)

Name

gamma, lgamma, signgam – log gamma function

Syntax

#include <math.h>

double gamma(*x***)**
double *x***;**

double lgamma(*x***)**
double *x***;**

extern int `signgam;`

Description

The `gamma` function returns ln $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer `signgam`. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0)
        error();
y = exp(y);
if(signgam)
        y = -y;
```

The `lgamma` function is another name for the gamma function.

Return Value

The `gamma` and `lgamma` functions return HUGE_VAL and set *errno* to EDOM when *x* is an integer value less than or equal to zero. When the correct value would overflow they return HUGE_VAL and set *errno* to ERANGE.

Environment

When your program is compiled using the System V environment for nonpositive integer values, HUGE is returned, and *errno* is set to EDOM. A message indicating DOMAIN error is printed on the standard error output.

If the correct value would overflow, `gamma` returns HUGE and sets *errno* to ERANGE.

These error-handling procedures may be changed with the function `matherr`(3m).

See Also

matherr(3m)

Name

hypot, cabs – Euclidean distance, complex absolute value

Syntax

#include <math.h>

double hypot(*x,y***)**
double *x,y;*

float fhypot(float x, float y)

double cabs(*z***)**
struct {double x,y;} *z***;**

float fcabs(*z***)**
struct {float x,y;} *z***;**

Description

The `hypot`, `fhypot`, `cabs`, and `fcabs` functions return the following:

```
sqrt(x*x+y*y)
```

This computation prevents underflows and overflows only if the final result dictates it.

The functions `fhypot` and `fcabs` are equivalent to the `hypot` and `cabs` function with the exception of float data type.

Error

When rounding off, for example, below 0.97 *ulps*. Consequently `hypot` (5.0,12.0) = 13.0 exactly; in general, `hypot` and `cabs` return an integer whenever an integer might be expected.

The same cannot be said for the shorter and faster version of `hypot` and `cabs` that is provided in the comments in cabs.c; its error can exceed 1.2 *ulps*.

Return Value

If the correct value overflows, `hypot` and `cabs` return HUGE_VAL. If *x* or *y* is *NaN*, then *NaN* is returned.

See Also

math(3m), sqrt(3m)

Name

hypot, cabs – Euclidean distance

Syntax

#include <math.h>

double hypot(*x,y***)**
double *x,y***;**

double cabs(*z***)**
struct {double x,y;} *z***;**

Description

The `hypot` and `cabs` functions return

sqrt(*x***x* + *y***y*),

taking precautions against unwarranted overflows.

Return Value

The `hypot` and `cabs` functions return HUGE_VAL and sets *errno* to ERANGE when the correct value would overflow.

Environment

When your program is compiled using the System V environment, if the correct value would overflow, `hypot` returns HUGE and sets *errno* to ERANGE.

These error-handling procedures may be changed with the function `matherr`(3m).

The `cabs` subroutine does not exist in the System V environment. For sqrt, see `exp`(3m).

See Also

exp(3m)

Name

copysign, drem, finite, logb, scalb – copysign, remainder, exponent manipulations

Syntax

```
#include <math.h>

double copysign(x,y)
double x,y;

double drem(x,y)
double x,y;

int finite(x)
double x;

double logb(x)
double x;

double scalb(x,n)
double x;
int n;
```

Description

These functions are required, or recommended by the IEEE standard 754 for floating–point arithmetic.

The `copysign` function returns x with its sign changed to y's.

The `drem(x,y)` function returns the remainder r := x – n*y where n is the integer nearest the exact value of x/y. Additionally if |n – x/y| = 1/2, then n is even. Consequently the remainder is computed exactly and |r| ≤ |y|/2. Note that `drem(x,0)` is the exception (see DIAGNOTICS).

Finite(x) = 1 just when –∞ < x < +∞,
= 0 otherwise (when |x| = ∞ or x is *NaN*)

The `logb(x)` returns a signed integer converted to double–precision floating–point and so chosen that 1 ≤ |x|/2**n < 2 unless x = 0 or |x| = ∞ or x lies between 0 and the Underflow Threshold.

Scalb(x,n) = x*(2**n) computed, for integer n, without first computing 2**N.

Diagnostics

IEEE 754 defines drem(x,0) and drem(∞,y) to be invalid operations that produce a *NaN*.

IEEE 754 defines logb(±∞) = +∞ and logb(0) = –∞, and requires the latter to signal Division–by–Zero.

Restrictions

IEEE 754 currently specifies that logb(denormalized no.) = logb(tiniest normalized no. > 0) but the consensus has changed to the specification in the new proposed IEEE standard p854, namely that logb(x) satisfy

1 ≤ scalb(|x|,–logb(x)) < Radix ... = 2 for IEEE 754

for every x except 0, ∞ and *NaN*. Almost every program that assumes 754's specification will work correctly if logb follows 854's specification instead.

IEEE 754 requires copysign(x,*NaN*) = ±x but says nothing else about the sign of a *NaN*.

See Also

floor(3M), fp_class(3), math(3M)

Name

isnand, isnanf – test for floating point NaN (Not-A-Number)

Syntax

#include <ieeefp.h>

int isnand (dsrc)
double dsrc;

int isnanf (fsrc)
float fsrc;

Description

The `isnand` and `isnanf` routines return the value 1 for true if the argument dsrc or fsrc is a NaN; otherwise they return the value 0 for false.

Neither routine generates any exception, even for signaling NaNs.

The `isnan` function is implemented as a macro included in <ieeefp.h>.

Name

math – introduction to mathematical library functions

Description

These functions constitute the C math library *libm*. There are two versions of the math library *libm.a* and *libm43.a*.

The first, *libm.a,* contains routines written in MIPS assembly language and tuned for best performance and includes many routines for the *float* data type. The routines in there are based on the algorithms of Cody and Waite or those in the 4.3 BSD release, whichever provides the best performance with acceptable error bounds. Those routines with Cody and Waite implementations are marked with a '*' in the list of functions below.

The second version of the math library, *libm43.a,* contains routines all based on the original codes in the 4.3 BSD release. The difference between the two version's error bounds is typically around 1 unit in the last place, whereas the performance difference may be a factor of two or more.

The link editor searches this library under the ''–lm'' (or ''–lm43'') option. Declarations for these functions may be obtained from the include file <*math.h*>. The Fortran math library is described in ''man 3f intro''.

List Of Functions

The cycle counts of all functions are approximate; cycle counts often depend on the value of argument. The error bound sometimes applies only to the primary range.

Name	Description	Error Bound (ULPs) libm.a	Error Bound (ULPs) libm43.a	Cycles libm.a	Cycles libm43.a
acos	inverse trig function	3	3	?	?
acosh	inverse hyperbolic function	3	3	?	?
asin	inverse trig function	3	3	?	?
asinh	inverse hyperbolic function	3	3	?	?
atan	inverse trig function	1	1	152	260
atanh	inverse hyperbolic function	3	3	?	?
atan2	inverse trig function	2	2	?	?
cabs	complex absolute value	1	1	?	?
cbrt	cube root	1	1	?	?
ceil	integer no less than	0	0	?	?
copysign	copy sign bit	0	0	?	?
cos*	trig function	2	1	128	243
cosh*	hyperbolic function	?	3	142	294
drem	remainder	0	0	?	?
erf	error function	?	?	?	?

erfc	complementary error function	?	?	?	?
exp*	exponential	2	1	101	230
expm1	exp(x)–1	1	1	281	281
fabs	absolute value	0	0	?	?
fatan*	inverse trig function	3		64	
fcos*	trig function	1		87	
fcosh*	hyperbolic function	?		105	
fexp*	exponential	1		79	
flog*	natural logarithm	1		100	
floor	integer no greater than	0	0	?	?
fsin*	trig function	1		68	
fsinh*	hyperbolic function	?		44	
fsqrt	square root	1		95	
ftan*	trig function	?		61	
ftanh*	hyperbolic function	?		116	
hypot	Euclidean distance	1	1	?	?
j0	bessel function	?	?	?	?
j1	bessel function	?	?	?	?
jn	bessel function	?	?	?	?
lgamma	log gamma function	?	?	?	?
log*	natural logarithm	2	1	119	217
logb	exponent extraction	0	0	?	?
log10*	logarithm to base 10	3	3	?	?
log1p	log(1+x)	1	1	269	269
pow	exponential x**y	60–500	60–500	?	?
rint	round to nearest integer	0	0	?	?
scalb	exponent adjustment	0	0	?	?
sin*	trig function	2	1	101	222
sinh*	hyperbolic function	?	3	79	292
sqrt	square root	1	1	133	133
tan*	trig function	?	3	92	287
tanh*	hyperbolic function	?	3	156	293
y0	bessel function	?	?	?	?
y1	bessel function	?	?	?	?
yn	bessel function	?	?	?	?

In 4.3 BSD, distributed from the University of California in late 1985, most of the foregoing functions come in two versions, one for the double–precision "D" format in the DEC VAX–11 family of computers, another for double–precision arithmetic conforming to the IEEE Standard 754 for Binary Floating–Point Arithmetic. The two versions behave very similarly, as should be expected from programs more accurate and robust than was the norm when UNIX was born. For instance, the programs are accurate to within the numbers of *ulp*s tabulated above; an *ulp* is one *U*nit in the *L*ast *P*lace. And the programs have been cured of anomalies that afflicted the older math library *libm* in which incidents like the following had been reported:

sqrt(–1.0) = 0.0 and log(–1.0) = –1.7e38.
cos(1.0e–11) > cos(0.0) > 1.0.
pow(x,1.0) ≠ x when x = 2.0, 3.0, 4.0, ..., 9.0.

pow(–1.0,1.0e10) trapped on Integer Overflow.
sqrt(1.0e30) and sqrt(1.0e–30) were very slow.

RISC machines conform to the IEEE Standard 754 for Binary Floating–Point Arithmetic, to which only the notes for IEEE floating-point apply and are included here.

BIEEE STANDARD 754 Floating–Point Arithmetic:

This standard is on its way to becoming more widely adopted than any other design for computer arithmetic.

The main virtue of 4.3 BSD's *libm* codes is that they are intended for the public domain; they may be copied freely provided their provenance is always acknowledged, and provided users assist the authors in their researches by reporting experience with the codes. Therefore no user of UNIX on a machine that conforms to IEEE 754 need use anything worse than the new *libm*.

Properties of IEEE 754 Double–Precision:

Wordsize: 64 bits, 8 bytes. **Radix:** Binary.
Precision: 53 significant bits, roughly like 16 significant decimals.
If x and x' are consecutive positive Double–Precision numbers (they differ by 1 *ulp*), then
1.1e–16 < 0.5**53 < (x'–x)/x ≤ 0.5**52 < 2.3e–16.
Range: Overflow threshold = 2.0**1024 = 1.8e308
Underflow threshold = 0.5**1022 = 2.2e–308
Overflow goes by default to a signed ∞.
Underflow is *Gradual,* rounding to the nearest integer multiple of 0.5**1074 = 4.9e–324.
Zero is represented ambiguously as +0 or –0.
Its sign transforms correctly through multiplication or division, and is preserved by addition of zeros with like signs; but x–x yields +0 for every finite x. The only operations that reveal zero's sign are division by zero and copysign(x,±0). In particular, comparison (x > y, x ≥ y, etc.) cannot be affected by the sign of zero; but if finite x = y then ∞ = 1/(x–y) ≠ –1/(y–x) = –∞.
∞ is signed.
it persists when added to itself or to any finite number. Its sign transforms correctly through multiplication and division, and (finite)/±∞ = ±0 (nonzero)/0 = ±∞. But ∞–∞, ∞*0 and ∞/∞ are, like 0/0 and sqrt(–3), invalid operations that produce *NaN*. ...
Reserved operands:
there are 2**53–2 of them, all called *NaN* (*N*ot *a N*umber). Some, called Signaling *NaN*s, trap any floating–point operation performed upon them; they could be used to mark missing or uninitialized values, or nonexistent elements of arrays. The rest are Quiet *NaN*s; they are the default results of Invalid Operations, and propagate through subsequent arithmetic operations. If x ≠ x then x is *NaN*; every other predicate (x > y, x = y, x < y, ...) is FALSE if *NaN* is involved.

NOTE

Trichotomy is violated by *NaN*. Besides being FALSE, predicates that entail ordered comparison, rather than mere (in)equality, signal Invalid Operation when *NaN* is involved.

Rounding:

Every algebraic operation (+, –, *, /, √) is rounded by default to within half an *ulp*, and when the rounding error is exactly half an *ulp* then the rounded value's least significant bit is zero. This kind of rounding is usually the best kind, sometimes provably so; for instance, for every x = 1.0, 2.0, 3.0, 4.0, ..., 2.0**52, we find (x/3.0)*3.0 == x and (x/10.0)*10.0 == x and ... despite that both the quotients and the products have been rounded. Only rounding like IEEE 754 can do that. But no single kind of rounding can be proved best for every circumstance, so IEEE 754 provides rounding towards zero or towards +∞ or towards –∞ at the programmer's option. And the same kinds of rounding are specified for Binary–Decimal Conversions, at least for magnitudes between roughly 1.0e–10 and 1.0e37.

Exceptions:

IEEE 754 recognizes five kinds of floating–point exceptions, listed below in declining order of probable importance.

Exception	Default Result
Invalid Operation	*NaN*, or FALSE
Overflow@±∞	
Divide by Zero	±∞
Underflow	Gradual Underflow
Inexact	Rounded value

NOTE

An Exception is not an Error unless handled badly. What makes a class of exceptions exceptional is that no single default response can be satisfactory in every instance. On the other hand, if a default response will serve most instances satisfactorily, the unsatisfactory instances cannot justify aborting computation every time the exception occurs.

For each kind of floating–point exception, IEEE 754 provides a Flag that is raised each time its exception is signaled, and stays raised until the program resets it. Programs may also test, save and restore a flag. Thus, IEEE 754 provides three ways by which programs may cope with exceptions for which the default result might be unsatisfactory:

1) Test for a condition that might cause an exception later, and branch to avoid the exception.

2) Test a flag to see whether an exception has occurred since the program last reset its flag.

3) Test a result to see whether it is a value that only an exception could have produced.

NOTE

The only reliable ways to discover whether Underflow has occurred are to test whether products or quotients lie closer to zero than the underflow threshold, or to test the Underflow flag. (Sums and differences cannot underflow in IEEE 754; if $x \neq y$ then $x-y$ is correct to full precision and certainly nonzero regardless of how tiny it may be.) Products and quotients that underflow gradually can lose accuracy gradually without vanishing, so comparing them with zero (as one might on a VAX) will not reveal the loss. Fortunately, if a gradually underflowed value is destined to be added to something bigger than the underflow threshold, as is almost always the case, digits lost to gradual underflow will not be missed because they would have been rounded off anyway. So gradual underflows are usually *provably* ignorable. The same cannot be said of underflows flushed to 0.

At the option of an implementor conforming to IEEE 754, other ways to cope with exceptions may be provided:

4) ABORT. This mechanism classifies an exception in advance as an incident to be handled by means traditionally associated with error–handling statements like "ON ERROR GO TO ...". Different languages offer different forms of this statement, but most share the following characteristics:

— No means is provided to substitute a value for the offending operation's result and resume computation from what may be the middle of an expression. An exceptional result is abandoned.

— In a subprogram that lacks an error–handling statement, an exception causes the subprogram to abort within whatever program called it, and so on back up the chain of calling subprograms until an error–handling statement is encountered or the whole task is aborted and memory is dumped.

5) STOP. This mechanism, requiring an interactive debugging environment, is more for the programmer than the program. It classifies an exception in advance as a symptom of a programmer's error; the exception suspends execution as near as it can to the offending operation so that the programmer can look around to see how it happened. Quite often the first several exceptions turn out to be quite unexceptionable, so the programmer ought ideally to be able to resume execution after each one as if execution had not been stopped.

6) ... Other ways lie beyond the scope of this document.

The crucial problem for exception handling is the problem of Scope, and the problem's solution is understood, but not enough manpower was available to implement it fully in time to be distributed in 4.3 BSD's *libm*. Ideally, each elementary function should act as if it were indivisible, or atomic, in the sense that ...

i) No exception should be signaled that is not deserved by the data supplied to that function.

ii) Any exception signaled should be identified with that function rather than with one of its subroutines.

iii) The internal behavior of an atomic function should not be disrupted when a calling program changes from one to another of the five or so ways of handling exceptions listed above, although the definition of the function may be correlated intentionally with exception handling.

Ideally, every programmer should be able *conveniently* to turn a debugged subprogram into one that appears atomic to its users. But simulating all three characteristics of an atomic function is still a tedious affair, entailing hosts of tests and saves–restores; work is under way to ameliorate the inconvenience.

Meanwhile, the functions in *libm* are only approximately atomic. They signal no inappropriate exception except possibly ...

Over/Underflow
when a result, if properly computed, might have lain barely within range, and

Inexact in *cabs*, *cbrt*, *hypot*, *log10* and *pow*
when it happens to be exact, thanks to fortuitous cancellation of errors.

Otherwise, ...

Invalid Operation is signaled only when
any result but *NaN* would probably be misleading.

Overflow is signaled only when
the exact result would be finite but beyond the overflow threshold.

Divide–by–Zero is signaled only when
a function takes exactly infinite values at finite operands.

Underflow is signaled only when
the exact result would be nonzero but tinier than the underflow threshold.

Inexact is signaled only when
greater range or precision would be needed to represent the exact result.

Exceptions on RISC machines:

The exception enables and the flags that are raised when an exception occurs (as well as the rounding mode) are in the floating–point control and status register. This register can be read or written by the routines described on the man page *fpc*(3). This register's layout is described in the file <*mips/fpu.h*> in UMIPS–BSD releases and in <*sys/fpu.h*> in UMIPS–SYSV releases.

What is currently available is only the raw interface which was only intended to be used by the code to implement IEEE user trap handlers. IEEE floating–point exceptions are enabled by setting the enable bit for that exception in the floating–point control and status register. If an exception then occurs the UNIX signal SIGFPE is sent to the process. It is up to the signal handler to determine the instruction that caused the exception and to take the action specified by the user. The instruction that caused the exception is in one of two places. If the floating–point board is used (the floating–point implementation revision register indicates this in

Name

asinh, acosh, atanh – inverse hyperbolic functions

Syntax

```
#include <math.h>

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;
```

Description

These functions compute the designated inverse hyperbolic functions for real arguments.

Return Value

The function `acosh` returns 0.0 if the argument is less than 1.

The function `atanh` returns the HUGE value if the argument has absolute value greater than or equal to 1.

See Also

exp(3m), intro(3m)

Name

j0, j1, jn, y0, y1, yn – bessel functions

Syntax

#include <math.h>

double j0(*x*)
double *x;*

double j1(*x*)
double *x;*

double jn(*n,x*)
double *x;*

double y0(*x*)
double *x;*

double y1(*x*)
double *x;*

double yn(*n,x*)
double *x;*

Description

These functions calculate bessel functions of the first and second kinds for real arguments and integer orders.

Return Value

Negative arguments cause `y0`, `y1`, and `yn` to return *NaN*. Arguments too large in magnitude cause `y0`, `y1`, and `yn` to return *NaN*.

Arguments too large in magnitude cause `j0`, `j1`, and `jn` to return zero.

Environment

When your program is compiled using the System V environment, nonpositive arguments cause `y0`, `y1` and `yn` to return the value HUGE and to set *errno* to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

Arguments too large in magnitude cause `j0`, `j1`, `y0`, and `y1` to return zero and to set *errno* to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output.

These error-handling procedures may be changed with the `matherr`(3m) function.

See Also

math(3m)

it's implementation field) then the instruction that caused the exception is in the floating–point exception instruction register. In all other implementations the instruction that caused the exception is at the address of the program counter as modified by the branch delay bit in the cause register. Both the program counter and cause register are in the sigcontext structure passed to the signal handler (see `signal`(3)). If the program is to be continued past the instruction that caused the exception the program counter in the signal context must be advanced. If the instruction is in a branch delay slot then the branch must be emulated to determine if the branch is taken and then the resulting program counter can be calculated (see `emulate_branch`(3) and `signal`(3)).

Restrictions

When signals are appropriate, they are emitted by certain operations within the codes, so a subroutine–trace may be needed to identify the function with its signal in case method 5) above is in use. And the codes all take the IEEE 754 defaults for granted; this means that a decision to trap all divisions by zero could disrupt a code that would otherwise get correct results despite division by zero.

See Also

fpc(3), signal(3), emulate_branch(3)
R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description

An explanation of IEEE 754 and its proposed extension p854 was published in the IEEE magazine MICRO in August 1984 under the title "A Proposed Radix– and Word–length–independent Standard for Floating–point Arithmetic" by W. J. Cody et al.

Articles in the IEEE magazine COMPUTER vol. 14 no. 3 (Mar. 1981), and in the ACM SIGNUM Newsletter Special Issue of Oct. 1979, may be helpful although they pertain to superseded drafts of the standard.

Name

matherr – error-handling function for System V math library

Syntax

#include <math.h>

int matherr (x)
struct exception *x;

Description

The `matherr` subroutine is invoked by functions in the System V Math Library when errors are detected. Users may define their own procedures for handling errors by including a function named `matherr` in their programs. The `matherr` subroutine must be of the form described above. A pointer to the exception structure *x* will be passed to the user-supplied `matherr` function when an error occurs. This structure, which is defined in the <math.h> header file, is as follows:

```
struct exception {
      int type;
      char *name;
      double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

```
DOMAIN      domain error
SING        singularity
OVERFLOW    overflow
UNDERFLOW   underflow
TLOSS       total loss of significance
PLOSS       partial loss of significance
```

The element *name* points to a string containing the name of the function that had the error. The variables *arg1* and *arg2* are the arguments to the function that had the error. The *retval* is a double that is returned by the function having the error. If it supplies a return value, the user's `matherr` must return nonzero. If the default error value is to be returned, the user's `matherr` must return 0.

If `matherr` is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to nonzero and the program continues.

Examples

```
matherr(x)
register struct exception *x;
{
      switch (x->type) {
      case DOMAIN:
      case SING: /* print message and abort */
            fprintf(stderr, "domain error in %s\n", x->name);
            abort( );
```

```
        case OVERFLOW:
                if (!strcmp("exp", x->name)) {
                        /* if exp, print message, return the argument */
                        fprintf(stderr, "exp of %f\n", x->arg1);
                        x->retval = x->arg1;
                } else if (!strcmp("sinh", x->name)) {
                        /* if sinh, set errno, return 0 */
                        errno = ERANGE;
                        x->retval = 0;
                } else
                        /* otherwise, return HUGE */
                        x->retval = HUGE;
                break;
        case UNDERFLOW:
                return (0); /* execute default procedure */
        case TLOSS:
        case PLOSS:
                /* print message and return 0 */
                fprintf(stderr, "loss of significance in %s\n", x->name);
                x->retval = 0;
                break;
        }
        return (1);
}
```

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
BESSEL:	–	–	H	0	M, 0	*
y0, y1, yn (neg. no.)	M, –H	–	–	–	–	–
EXP:	–	–	H	0	–	
POW:	–	–	H	0	–	–
(neg.)**(non-int.), 0**0	M, 0	–	–	–	–	–
LOG:						
log(0):	–	M, –H	–	–	–	–
log(neg.):	M, –H	–	–	–	–	–
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	–	–	–	–
HYPOT:	–	–	H	–	–	–
SINH, COSH:	–	–	H	–	–	–
SIN, COS:	–	–	–	–	M, 0	*
TAN:	–	–	H	–	M, 0	*
ACOS, ASIN:	M, 0	–	–	–	–	–

ABBREVIATIONS

*	As much as possible of the value is returned.
M	Message is printed.
H	HUGE is returned.
–H	–HUGE is returned.
0	0 is returned.

Name

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions and their inverses

Syntax

#include <math.h>

double sin(*x***)**
double *x***;**

float fsin(*x***)**
float *x***;**

double cos(*x***)**
double *x***;**

float fcos(*x***)**
float *x***;**

double tan(*x***)**
double *x***;**

float ftan(*x***)**
float *x***;**

double asin(*x***)**
double *x***;**

float fasin(*x***)**
float *x***;**

double acos(*x***)**
double *x***;**

float facos(*x***)**
float *x***;**

double atan(*x***)**
double *x***;**

float fatan(*x***)**
float *x***;**

double atan2(*y,x***)**
double *y,x***;**

float fatan2(*y,x***)**
float *y,x***;**

Description

The `sin`, `cos`, and `tan` functions return trigonometric functions of radian arguments *x* for double data types.

The `fsin`, `fcos`, and `ftan` functions return trigonometric functions for float data types.

The `asin` and `fasin` functions return the arc sine in the range $-\pi/2$ to $\pi/2$ for double and float data types, respectively.

The `acos` and `facos` functions return the arc cosine in the range 0 to π for double and float data types, respectively.

The `atan` and `fatan` functions return the arc tangent in the range $-\pi/2$ to $\pi/2$ for double and float data types, respectively.

The `atan2` and `fatan2` functions return the arc tangent of y/x in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value for double and float data types, respectively.

Error (due to roundoff)

When *P* stands for the number stored in the computer in place of π = 3.14159 26535 89793 23846 26433 ... . and "trig" stands for one of "sin", "cos" or "tan", then the expression "trig(x)" in a program actually produces an approximation to trig(x*π/P), and "atrig(x)" approximates (P/π)*atrig(x). The approximations are close.

P differs from π by a fraction of an *ulp*; the difference is apparent only if the argument x is huge, and even then the difference is likely to be swamped by the uncertainty in x. Every trigonometric identity that does not involve π explicitly is satisfied equally well regardless of whether P = π. For example, $\sin^2(x)+\cos^2(x) = 1$ and $\sin(2x) = 2\sin(x)\cos(x)$ to within a few *ulps* regardless of how big x is. Therefore, the difference between P and π is unlikely to effect scientific and engineering computations.

Return Value

All the double functions return *NaN* if *NaN* is passed in.

If |x| > 1 then `asin` (x) and `acos` (x) will return the default quiet *NaN*.

The `atan2` function defines `atan2` (0,0) = *NaN*.

See Also

hypot(3m), math(3m), sqrt(3m)

Name

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

Syntax

#include <math.h>

double sin(*x***)**
double *x***;**

double cos(*x***)**
double *x***;**

double tan(*x***)**
double *x***;**

double asin(*x***)**
double *x***;**

double acos(*x***)**
double *x***;**

double atan(*x***)**
double *x***;**

double atan2(*x,y***)**
double *x,y***;**

Description

The subroutines `sin, cos` and `tan,` return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

The `asin` subroutine returns the arc sin in the range $-\pi/2$ to $\pi/2$.

The `acos` subroutine returns the arc cosine in the range 0 to π.

The `atan` subroutine returns the arc tangent of *x* in the range $-\pi/2$ to $\pi/2$.

The `atan2` subroutine returns the arc tangent of *x*/*y* in the range $-\pi$ to π.

Restrictions

The value of `tan` for arguments greater than about 2**31 is unreliable.

Return Value

Arguments of magnitude greater than 1 cause `asin` and `acos` to return zero and set *errno* to EDOM.

The `atan2` function returns zero and sets *errno* to EDOM when *x* and *y* are both zero.

Environment

When your program is compiled using the System V environment, `sin, cos` and `tan` lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of

significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to ERANGE.

The `tan` subroutine returns HUGE for an argument which is near an odd multiple of $\pi/2$ when the correct value would overflow, and sets *errno* to ERANGE.

Arguments of magnitude greater than 1.0 cause `asin` and `acos` to return 0 and to set *errno* to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function `matherr`(3m).

Name

sinh, cosh, tanh – hyperbolic functions

Syntax

```
#include <math.h>

double sinh(x)
double x;

float fsinh(x)
float x;

double cosh(x)
double x;

float fcosh(x)
float x;

double tanh(x)
double x;

float ftanh(x)
float x;
```

Description

These functions compute the designated hyperbolic functions for double and float data types.

Error

Below 2.4 *ulps* (unit in the last place).

Diagnostics

The `sinh` and `cosh` functions return $+\infty$ (and *sinh* may return $-\infty$ for negative x) if the correct value would overflow.

See Also

math(3m)

Name

sinh, cosh, tanh – hyperbolic functions

Syntax

#include <math.h>

double sinh(*x***)**

double cosh(*x***)**
double *x***;**

double tanh(*x***)**
double *x***;**

Description

These functions compute the designated hyperbolic functions for real arguments.

Return Value

The `sinh` and `cosh` functions return HUGE_VAL and set *errno* to ERANGE when the correct value would overflow.

Environment

When your program is compiled using the System V environment, `sinh` and `cosh` return HUGE (and `sinh` may return HUGE or negative *x*) when the correct value would overflow and set *errno* to ERANGE.

These error-handling procedures may be changed with the function `matherr`(3m).

Name

cbrt, sqrt – cube root, square root

Syntax

#include <math.h>

double cbrt(x**)**
double x**;**

double sqrt(x**)**
double x**;**

float fsqrt(float x**)**
float x**;**

Description

The `cbrt` function returns the cube root of x.

The `sqrt` and `fsqrt` functions return the square root of x for double and float data types respectively.

Error Due to Roundoff and Other Reasons

The `cbrt` function is accurate to within 0.7 *ulp*s.

The `sqrt` function on this machine conforms to IEEE 754 and is correctly rounded in accordance with the rounding mode in force; the error is less than half an *ulp* in the default mode (round–to–nearest). An *ulp* is one *U*nit in the *L*ast *P*lace carried.

Diagnostics

The `sqrt` function returns the default quiet *NaN* when x is negative indicating the invalid operation.

See Also

math(3m)

Name

intro – introduction to mathematical library functions

Description

These functions constitute the math library, *libm*. They are automatically loaded as needed by the FORTRAN compiler `f77`(1). The link editor searches this library under the ''–lm'' option. Declarations for these functions may be obtained from the include file <math.h>.

VAX Only

On VAX machines, the GFLOAT version of *libm* is used when you use the `ld`(1) command with the **lcg** option. Note that you must use the GFLOAT version of *libm* with modules compiled using the `cc`(1) with the **–Mg** option.

Also on VAX machines, note that neither the compiler nor the linker `ld`(1) can detect when mixed double floating point types are used, and the program may produce erroneous results if this occurs.

System V Compatibility

This library contains System V compatibility features that are available to general ULTRIX programs. For a discussion of how these features are documented, and how to specify that the System V environment is to be used in compiling and linking your programs, see `intro`(3).

Files

/usr/lib/libma
/usr/lib/libmg.a (VAX only)

Name

asinh, acosh, atanh – inverse hyperbolic functions

Syntax

#include <math.h>

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;

Description

The `asinh`, `acosh`, and `atanh` functions compute the designated inverse hyperbolic functions for real arguments.

Errors Because of Roundoff, Etc.

These functions inherit much of their error from the `log1p`(3m) function.

Diagnostics

The `acosh` function returns the default quiet *NaN* if the argument is less than one.

The `atanh` function returns the default quiet *NaN* if the argument has an absolute value greater than or equal to one.

See Also

exp(3m), math(3m)